Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.067536

REVIEW

Towards Secure APIs: A Survey on RESTful API Vulnerability Detection

Fatima Tanveer[1], **Faisal Iradat**[1,*], **Waseem Iqbal**[2,*] **and Awais Ahmad**[3]

[1]Department of Computer Science, School of Mathematics and Science, Institute of Business Administration, Karachi, 75270, Pakistan

[2]Department of Electrical and Computer Engineering, Sultan Qaboos University, Al-Khud, Muscat, 123, Oman

[3]College of Computer and Information Sciences, Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, 11432, Saudi Arabia

*Corresponding Authors: Faisal Iradat. Email: firadat@iba.edu.pk; Waseem Iqbal. Email: m.waseem@squ.edu.om

Received: 06 May 2025; Accepted: 24 June 2025; Published: 30 July 2025

ABSTRACT: RESTful APIs have been adopted as the standard way of developing web services, allowing for smooth communication between clients and servers. Their simplicity, scalability, and compatibility have made them crucial to modern web environments. However, the increased adoption of RESTful APIs has simultaneously exposed these interfaces to significant security threats that jeopardize the availability, confidentiality, and integrity of web services. This survey focuses exclusively on RESTful APIs, providing an in-depth perspective distinct from studies addressing other API types such as GraphQL or SOAP. We highlight concrete threats—such as injection attacks and insecure direct object references (IDOR)—to illustrate the evolving risk landscape. Our work systematically reviews state-of-the-art detection methods, including static code analysis and penetration testing, and proposes a novel taxonomy that categorizes vulnerabilities such as authentication and authorization issues. Unlike existing taxonomies focused on general web or network-level threats, our taxonomy emphasizes API-specific design flaws and operational dependencies, offering a more granular and actionable framework for RESTful API security. By critically assessing current detection methodologies and identifying key research gaps, we offer a structured framework that advances the understanding and mitigation of RESTful API vulnerabilities. Ultimately, this work aims to drive significant advancements in API security, thereby enhancing the resilience of web services against evolving cyber threats.

KEYWORDS: RESTful API; vulnerability detection; API security; taxonomy; systematic review

1 Introduction

Representational State Transfer Application Programming Interfaces (RESTful APIs) have been embraced as the standard web service architecture. This architecture follows a stateless protocol that allows clients to send requests to servers over Hypertext Transfer Protocol (HTTP) and Hypertext Transfer Protocol Secure (HTTPS) using methods like GET, POST, PUT, and DELETE [1]. These APIs are integral to modern web architectures due to their simplicity, scalability, and compatibility with a wide range of platforms and devices [2]. The principles put forward in the RESTful architecture allow developers to build APIs that are simple to integrate and utilize, thus making APIs easily deployable in different contexts [3,4].

However, along with these advantages, RESTful APIs face numerous security threats. Their accessibility over the Internet makes them attractive targets for attackers. The stateless nature of RESTful APIs requires each client request to contain all necessary information for processing, which, if not properly secured, can inadvertently leak sensitive data [3,5].

Recent statistics underscore the critical need for enhanced API security. According to the 2023 State of API Security Report, 60% of organizations reported a data breach in the past two years. Of these, 74% experienced at least three API-related breaches, with 40% facing five or more, and 11% encountering over seven, highlighting the urgency for robust API security measures [6]. Additionally, reports indicate that 95% of companies have had an API security incident in the past 12 months, with API attack traffic growing by 681% [7].

The OWASP API Security Top 10 (2023) report identifies the most critical API security risks, including:

1. **Broken Object Level Authorization (BOLA):** Improper authorization checks allowing unauthorized access to objects.
2. **Broken Authentication:** Flaws in authentication mechanisms leading to unauthorized access.
3. **Excessive Data Exposure:** APIs exposing more data than necessary, increasing the attack surface.
4. **Lack of Resources and Rate Limiting:** Absence of controls to limit the number of requests, leading to potential abuse.
5. **Broken Function Level Authorization:** Insufficient authorization checks at the function level, allowing unauthorized actions.
6. **Mass Assignment:** Binding client-provided data (e.g., JSON) to data models without proper filtering, leading to unauthorized data manipulation.
7. **Security Misconfiguration:** Improper configuration of security settings, leaving APIs vulnerable.
8. **Injection:** Injection flaws (e.g., SQL, NoSQL) allowing attackers to execute malicious commands.
9. **Improper Assets Management:** Lack of inventory and management of API hosts, leading to exposure of deprecated or vulnerable API versions.
10. **Insufficient Logging and Monitoring:** Failure to log and monitor API activities, hindering detection of security incidents [8].

Visualizing how RESTful API threats work makes the problem more tangible for the reader. Below are examples of common API vulnerabilities:

- **Insecure Direct Object Reference (IDOR):** Attackers can manipulate object references to access unauthorized data by altering URL parameters.
- **SQL Injection (SQLi):** Attackers inject malicious SQL code through unsanitized inputs to manipulate databases.
- **Cross-Site Scripting (XSS):** Attackers inject malicious scripts into web pages viewed by other users, leading to unauthorized actions.

Understanding these attack vectors is crucial for implementing effective security measures to protect RESTful APIs from potential threats. As illustrated in Fig. 1, different types of vulnerabilities emerge at various layers of the RESTful API architecture.

Scope of This Survey

This work **exclusively targets RESTful APIs**, providing an in-depth analysis of their security challenges and solutions. We do not extend our review to alternative API architectures such as GraphQL, gRPC, or SOAP. The focus is to address the unique risks and vulnerabilities inherent to RESTful API design and deployment.

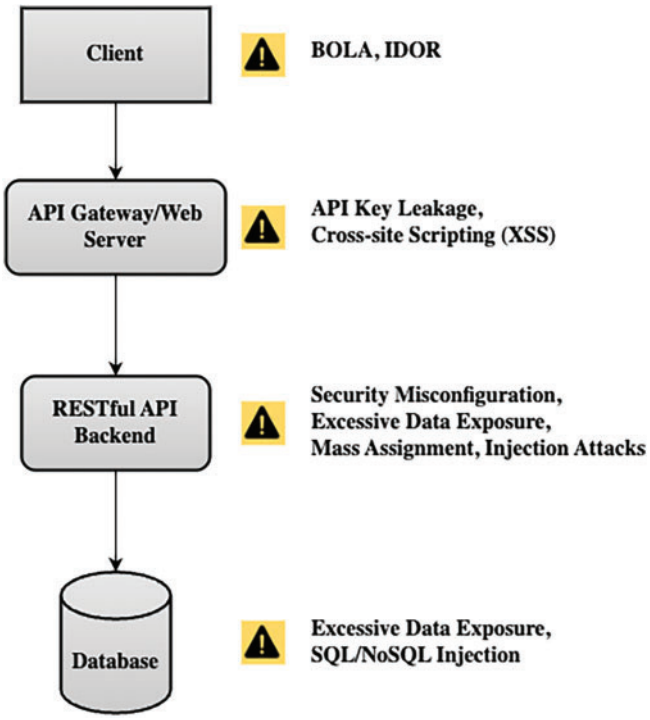

Figure 1: RESTful API architecture with mapped vulnerabilities

1.1 Rationale for the Study

RESTful APIs have become crucial in contemporary web services due to their widespread use in simplifying interactions between clients and servers [6,9]. Their popularity stems from ease of use, flexibility, and adaptability to different systems. However, this widespread adoption has also made these APIs vulnerable to numerous cyber threats, impacting the availability, confidentiality, and integrity of web services. The integral role of RESTful APIs in enabling essential processes means that their security risks cannot be ignored [6,10].

Given the increasing complexity of cyber threats, it's imperative to regularly assess and enhance vulnerability detection methods. This paper aims to improve the security of RESTful APIs by exploring existing literature for gaps and proposing a novel taxonomy. By doing so, we aim to bolster the protection of these APIs against emerging threats and support the reliability and security of modern web services. This is essential for preserving the credibility and operability of web-based systems in today's expanding digital environment.

1.2 Research Problem

Despite the availability of numerous frameworks and tools for vulnerability detection in RESTful APIs, there are still many limitations in their practical deployment and effectiveness in the real world. Existing approaches tend to focus on particular vulnerability types or use limited test scenarios, which fail to capture the complexity and dynamism of real world APIs [1–3]. Other studies further reinforce this point [11–13]. However, this fragmented research paradigm impedes the development of a comprehensive methodology for evaluating the overall effectiveness of API vulnerability detection techniques.

In addition, APIs are inherently dynamic, and they are constantly changing, introducing new vulnerabilities or rendering existing detection mechanisms obsolete. Real world API implementations often differ from documented specifications, leading to deviations that obscure potential security threats or reduce detection accuracy, a critical yet insufficiently addressed challenge [13,14].

Furthermore, existing detection tools and techniques are insufficient in handling the complex relationships between API operations. Such dependencies are complex and context specific, and are difficult to capture effectively using only conventional static analysis techniques [3,15,16].

Considering the substantial research gaps in this area, it is imperative to develop enhanced current detection methods and advanced, adaptive solutions to address the changing landscape of RESTful API security threats. To address this need, our study performs a systematic literature review by integrating existing knowledge, thoroughly evaluating existing detection approaches, identifying major research gaps, and proposing robust and applicable to real world solutions to improve future RESTful API security practices [17–19].

To contextualize our contributions, Table 1 provides a comparison of key API security surveys and frameworks.

Table 1: Comparison of existing works and our study. Adapted from [6,7,9,16,20–22]

Category	Study	Scope	Methodology and limitations addressed in our work
Industry reports	Traceable AI report (2023) [6]	Offers descriptive analysis without structured vulnerability categorization	Provides structured classification along with empirical validation, significantly enhancing clarity and practical application.
	Salt security report (2024) [7]	Focuses on industry case studies without academic rigor or structured detection framework	We add academic rigor and structured methodologies, providing comprehensive vulnerability categorization supported by empirical validation.
	OWASP API security (2023) [9]	Broad classification based on common industry vulnerabilities, lacks hierarchical taxonomy	Our work introduces a detailed hierarchical taxonomy validated against real-world breaches, surpassing descriptive classification.
Empirical studies	Mazidi et al. (2024) [16]	Specific focus on Mass Assignment vulnerabilities, limited broader API security context	Extends analysis beyond Mass Assignment to comprehensive coverage of diverse RESTful API-specific threats.
	Wang et al. (2023) [20]	Fuzz testing approach, isolated from broader API security frameworks	Integrates fuzzing methods within a broader structured framework and real-world empirical scenarios.

(Continued)

Table 1 (continued)

Category	Study	Scope	Methodology and limitations addressed in our work
	Kim et al. (2023) [21]	Advanced NLP-driven testing methods without comprehensive vulnerability taxonomy integration	Incorporates NLP approaches into a comprehensive and structured vulnerability taxonomy, enhancing practical utility and validation.
	Liao et al. (2024) [22]	Empirical security risks study, limited specifically to general API ecosystems	Expands empirical analysis explicitly for RESTful API threats, including novel threats and structured detection techniques.
Our work	**This study**	Combines hierarchical taxonomy, comprehensive detection techniques, and empirical validation against industry data	Provides holistic vulnerability coverage, structured categorization, integrates latest industry trends (2023–2024), and addresses gaps identified in prior literature through empirical studies and structured analysis.

Intended Audience

This research is intended for several key audiences:

- **Cybersecurity Researchers and Academics:** To provide a foundation for further studies and development in the field of API security.
- **API Developers and Engineers:** To help them understand current vulnerabilities and improve security measures in their development practices.
- **IT Security Professionals:** To enhance their strategies and tools for detecting and mitigating vulnerabilities in RESTful APIs.
- **Organizations and Enterprises:** To inform their security policies and practices, ensuring the protection of their digital infrastructure.

1.3 Contributions

Given the significant gaps identified in prior research and the practical challenges associated with RESTful API security, this paper makes several substantial contributions aimed at advancing the field:

- **Systematic Literature Review:** We provide a comprehensive and systematic review of existing research on RESTful API vulnerability detection, evaluating both theoretical frameworks and practical tools. This review critically assesses the strengths and limitations of current methodologies, thereby facilitating future developments in this domain.

- **Hierarchical Taxonomy of API Vulnerabilities:** We introduce a structured, hierarchical taxonomy designed to classify RESTful API security risks, distinguishing clearly between input validation flaws, authentication weaknesses, and infrastructure misconfigurations. This structured classification deepens the understanding of vulnerability characteristics, facilitating targeted detection and mitigation strategies.
- **Bridging Theoretical Surveys and Empirical Data:** While existing industry-level insights provided by OWASP and Salt Security offer broad trends, our work specifically validates our proposed taxonomy with empirical real-world API breach data, thus ensuring the practical applicability and relevance of our research findings.
- **Identification of RESTful API-Specific Attack Surfaces:** We uniquely analyze vulnerabilities exclusive to RESTful APIs that traditional web security models often overlook, such as Mass Assignment, Automated Scraping, and API Token Hijacking. By addressing these previously underrepresented threats, our research significantly enhances existing knowledge on API-specific security concerns.
- **Alignment with Contemporary Industry Reports (2023–2024):** Our research incorporates recent attack trends and emerging threats reported in the latest industry security analyses (2023–2024), ensuring that our taxonomy remains timely, relevant, and aligned with current security landscapes and practices.
- **Identification of Key Research Gaps:** Through an extensive literature review, we explicitly identify research gaps that remain insufficiently addressed, including limited real-world testing scenarios, inadequate methods for detecting complex API dependencies, and insufficient handling of dynamically evolving APIs.
- **Future Research Directions:** We outline promising directions for future investigations that could enhance RESTful API security practices. These include refining input generation methods, advancing dependency detection algorithms, automating comprehensive API specification analyses, and validating vulnerability detection methods using real-world scenarios and datasets.

We synthesize these contributions to offer a robust and complete framework for understanding, categorizing, and addressing RESTful API security risks beyond what has previously been available in the fragmented methodologies. This comprehensive overview is intended to be a foundational reference for researchers and practitioners alike, and we expect it will help to develop more effective, robust security measures and to further strengthen the security posture of today's web services and APIs. Based on the insights presented here, future research can continue to refine detection and prevention techniques to respond to new threats.

A promising direction for future research is the integration of software-defined networking (SDN) and adaptive security architectures to better protect APIs in highly dynamic environments such as IoT. Recent research has demonstrated that SDN-based solutions can address many of the pressing security challenges in IoT deployments, highlighting the value of programmability and centralized control for mitigating evolving API threats [23].

1.4 Organization of the Paper

The remainder of this paper is structured as follows: Section 2 presents a systematic review of existing research on RESTful API vulnerabilities, critically evaluating both theoretical frameworks and practical tools, thus setting the stage for identifying research gaps. In Section 3, we describe the methodological approach employed in our review, including data collection, screening processes, and selection criteria, following the PRISMA guidelines. Section 4 proposes a comprehensive and hierarchical taxonomy for categorizing

RESTful API vulnerabilities, clearly distinguishing between authentication and authorization issues, data validation flaws, and configuration and deployment issues. Section 5 discusses common input sanitization methods and routine functions essential to mitigating API vulnerabilities. Subsequently, Section 6 highlights significant research gaps identified through our review, emphasizing areas that require further attention. In Section 7, we validate our taxonomy and methods using empirical data from real-world case studies, ensuring practical relevance. Section 8 offers a brief discussion given the practical considerations noted in the course of this research. Section 9 provides insightful directions for future research to advance the field of RESTful API security. Finally, Section 10 summarizes the key findings and contributions of this paper, outlining the implications for both researchers and practitioners in cybersecurity.

2 Related Work

Numerous studies have investigated the security landscape of RESTful APIs. The OWASP API Security Top 10 is widely regarded as the industry benchmark for identifying and categorizing API security threats [8]. This resource provides high-level insights into common vulnerabilities but does not offer a structured taxonomy or an empirical validation of attack trends. Similarly, commercial reports such as the Salt Security API Security Report (2024) and the Traceable AI API Threat Report (2023) document real-world attack trends and highlight the increasing frequency of API-related breaches [6,7]. While these reports provide valuable insights, they lack academic rigor, structured threat classification, and reproducible methodologies. To address this gap, Fig. 2 provides a synthesized heatmap that combines the OWASP API Top 10 (2023) ranking with estimated frequency data from industry reports. Recent systematic overviews of API misuse patterns [24] complement existing payload collections.

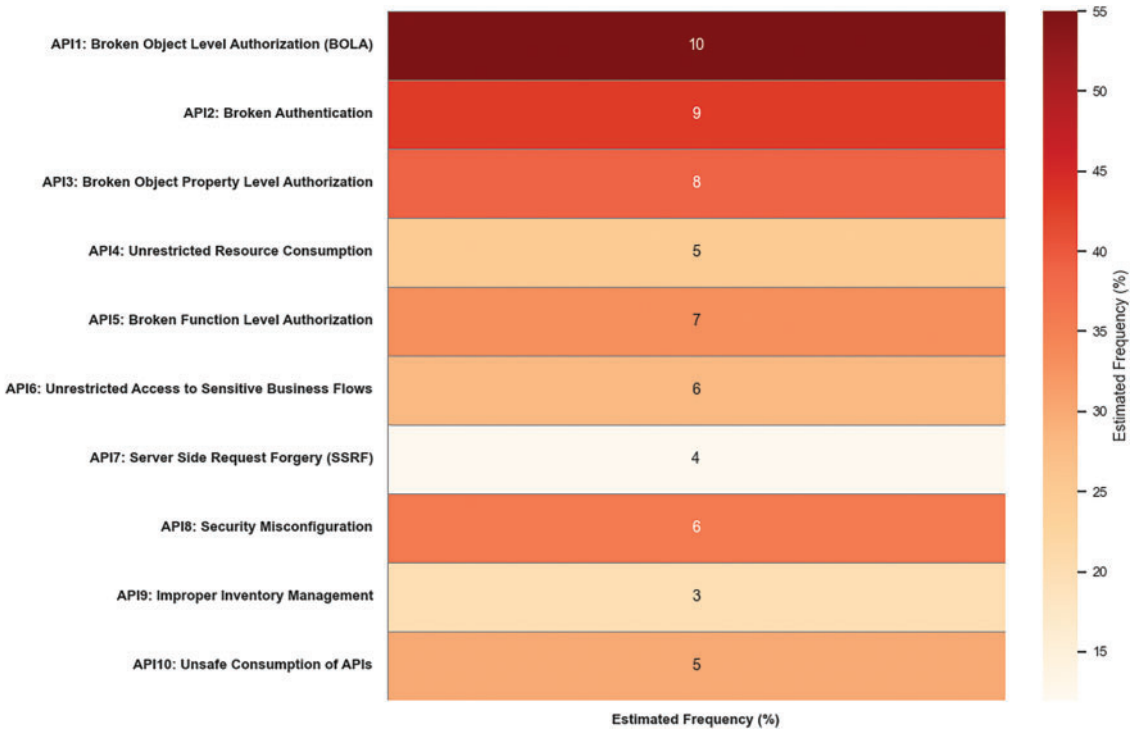

Figure 2: OWASP API Top 10 vulnerabilities heatmap. Adapted from [6–8]

Academic studies on RESTful API vulnerabilities can be categorized into three major areas:

1. Studies that classify API threats but lack a comprehensive taxonomy.
2. Detection-focused studies that emphasize scanning techniques without a systematic classification.
3. Empirical studies on API security, but with limited real-world validation.

Our work advances the field by combining all three—proposing a novel taxonomy, integrating detection strategies, and validating findings using real-world data.

Despite extensive literature on API security, critical gaps remain unaddressed:

1. **Lack of a Taxonomical Structure for API Threats:** Prior studies, such as Golmohammadi et al. [10], provide general classifications of web vulnerabilities but do not focus on RESTful API-specific attack surfaces. OWASP's API Security Top 10 categorizes API risks, but its classification is not hierarchical or structured to reflect the interdependencies between threats. Our taxonomy bridges this gap by structuring vulnerabilities into distinct categories based on impact, exploitation vectors, and detection complexity.
2. **Inadequate Discussion of API-Specific Attack Surfaces:** Most security surveys treat API threats as a subset of web application security, overlooking API-exclusive risks such as:
   - Broken Object Level Authorization (BOLA)
   - Mass Assignment
   - Improper API Asset Management
   - Business Logic Exploits

These threats do not fit into traditional web vulnerability categories, making them difficult to detect using existing methodologies [25].

3 Methodology

3.1 Systematic Review Process

We followed the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) guidelines to structure our systematic review.

Fig. 3 provides an overview of the flow of information through the different phases of the review.

To ensure a comprehensive analysis of API vulnerabilities, we conducted a systematic review following a well-defined and structured approach. In addition to traditional vulnerability datasets, we incorporated well-documented repositories such as the Rsnake XSS Cheat Sheet and the HTML5 Security Cheat Sheet which have been widely referenced in API security research [26,27]. Additionally, sources like the PortSwigger XSS Cheat Sheet [28], and the @XssPayloads Twitter account provides continuously updated payloads reflecting real-world security threats [29].

We focused on studies published between 2019 and 2024 because the landscape of RESTful API security has evolved significantly in recent years. The rise of GraphQL, API gateways, and zero-trust architectures has introduced new security challenges that older studies (pre-2019) do not adequately address. Additionally, recent industry reports (e.g., OWASP API Security Top 10) have identified new vulnerability classes that were previously unreported, necessitating a focus on contemporary research.

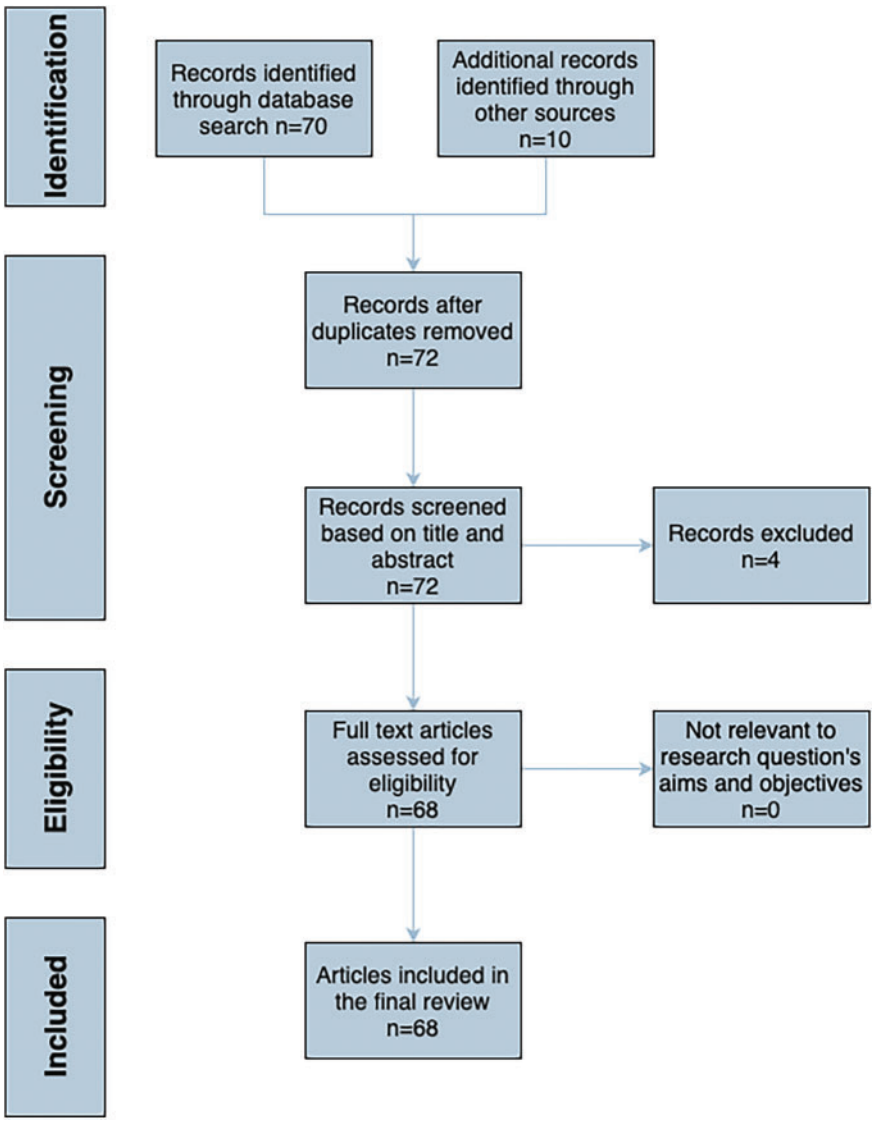

Figure 3: PRISMA Flow Diagram illustrating the selection process for studies included in the systematic review

As shown in Fig. 4, the proposed vulnerability detection pipeline consists of four key stages: API discovery, input generation, execution & monitoring, and vulnerability logging.

Fig. 5 gives an overview of the proposed approach to detect vulnerabilities in RESTful APIs. This flowchart outlines the sequential steps from initial API analysis to the final vulnerability mitigation, following the model proposed in [2].

3.2 Eligibility Criteria

To ensure the quality and relevance of the selected studies, we applied the following eligibility criteria:

- **Inclusion Criteria:** Studies that focus on RESTful API vulnerability detection, provide empirical validation, and present novel approaches, tools, or frameworks.
- **Exclusion Criteria:** Studies without empirical validation, those focusing on non-RESTful APIs, and duplicate or extended versions of previously published papers.

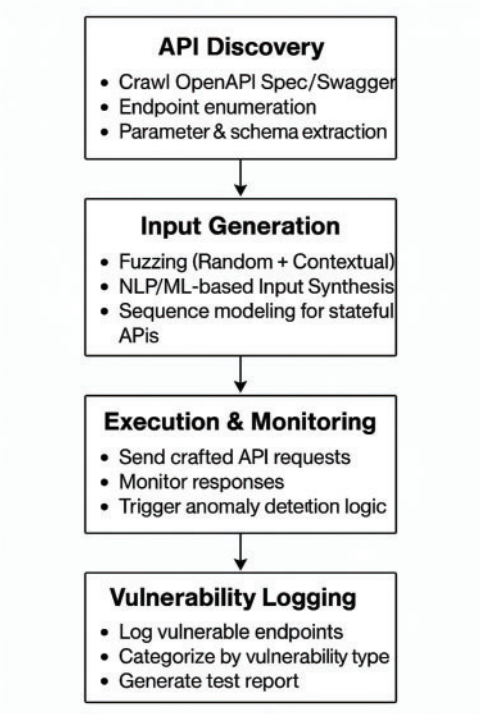

Figure 4: Flowchart illustrating the **RESTful** API vulnerability detection process from API discovery to vulnerability logging

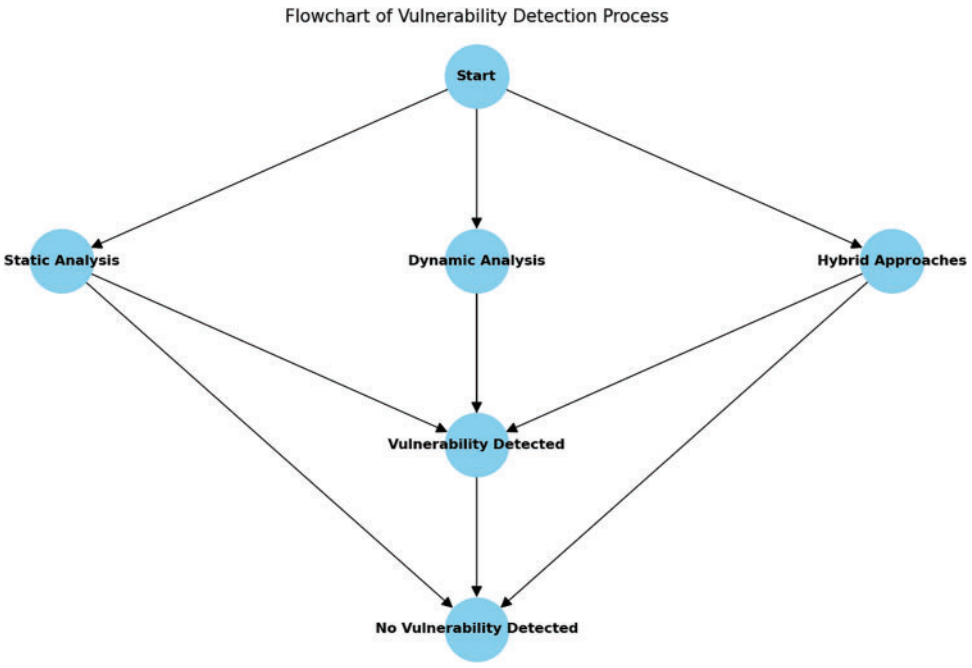

Figure 5: Flowchart illustrating the steps involved in the vulnerability detection process for RESTful APIs. Adapted from [2]

3.3 Information Sources

We utilized the following databases to gather a broad spectrum of research papers:

- Semantic Scholar: Popular for its coverage of computer science literature.
- Web of Science: A bibliographic database of scholarly articles from 22,000 peer-reviewed journals worldwide.
- IEEE Xplore: Selected for its coverage of engineering and technology articles.
- ACM Digital Library: A key database for computing and information technology research.

We selected IEEE Xplore, ACM Digital Library, and Semantic Scholar as our primary databases due to their strong coverage of peer-reviewed cybersecurity and software engineering research. Additionally, we included arXiv preprints to incorporate recent advancements in API security, particularly in federated learning and AI-driven threat detection.

3.4 Search Strategy

We designed tailored search queries to filter relevant studies, ensuring that we captured a comprehensive set of papers addressing RESTful API vulnerability detection. An example query used in our search process was:

Example Query: ("RESTful API" AND "vulnerability detection") OR ("API security" AND "dynamic analysis") AND ("IDOR" OR "BOLA")

3.5 Selection Process

The methods used to decide whether a study met the inclusion criteria of the review involved multiple reviewers screening each record and each report retrieved. The reviewers worked independently to ensure unbiased selection. Automation tools were used to manage the large volume of search results.

Our initial query retrieved 80 research papers across IEEE Xplore, ACM Digital Library, Semantic Scholar, and Web of Science, as well as additional resources. After title and abstract screening, we shortlisted 72 papers based on relevance to RESTful API security. A full-text review led to the inclusion of 68 papers in our final dataset, ensuring a focus on studies that provided empirical validation, novel detection methodologies, or a structured analysis of API threats.

3.6 Data Collection Process

The data collection process involved multiple reviewers independently extracting data from each report. Processes for obtaining or confirming data from study investigators were also utilized. Automation tools were employed to enhance the efficiency of data collection.

3.7 Data Items

We sought data for the following outcomes:

- Vulnerabilities identified in RESTful APIs.
- Methods used for vulnerability detection.
- Effectiveness of the detection methods.

3.8 Study Risk of Bias Assessment

The methods used to assess risk of bias in the included studies involved using standardized tools. Multiple reviewers assessed each study independently to ensure objectivity.

3.9 Effect Measures

For each outcome, we used effect measures such as risk ratios and mean differences to synthesize and present the results.

3.10 Synthesis Methods

The processes used to decide which studies were eligible for each synthesis involved tabulating the study intervention characteristics and comparing them against the planned groups for each synthesis.

3.11 Systematic Literature Review Overview

To ensure comprehensive and reproducible coverage, our literature review followed PRISMA guidelines, using explicit inclusion and exclusion criteria (see Section 3.2). We systematically searched four major academic databases and recent industry reports for studies published between 2019 and 2024, focusing on RESTful API vulnerability detection.

Table 2 summarizes the main sources and number of papers included in the review.

Table 2: Summary of systematic review

Database/Source	No. of papers included
IEEE Xplore	24
ACM digital library	16
Semantic scholar	12
Web of science	11
Industry reports (OWASP, Salt, Traceable)	5
Total included	68

4 Taxonomy of RESTful API Vulnerabilities

Based on our systematic review, we propose a novel taxonomy categorizing RESTful API vulnerabilities into the following categories: Authentication and Authorization, Data Validation Flaws, and Configuration and Deployment Issues. This taxonomy provides a structured framework for understanding the various types of vulnerabilities that can affect RESTful APIs and the methods used to detect them.

To systematically classify RESTful API vulnerabilities, we propose a hierarchical taxonomy, illustrated in Fig. 6. This taxonomy categorizes security threats into three major groups:

1. Authentication & Authorization Issues
2. Data Validation Flaws
3. Configuration & Deployment Issues

Each category is further divided into specific vulnerabilities, as outlined below.

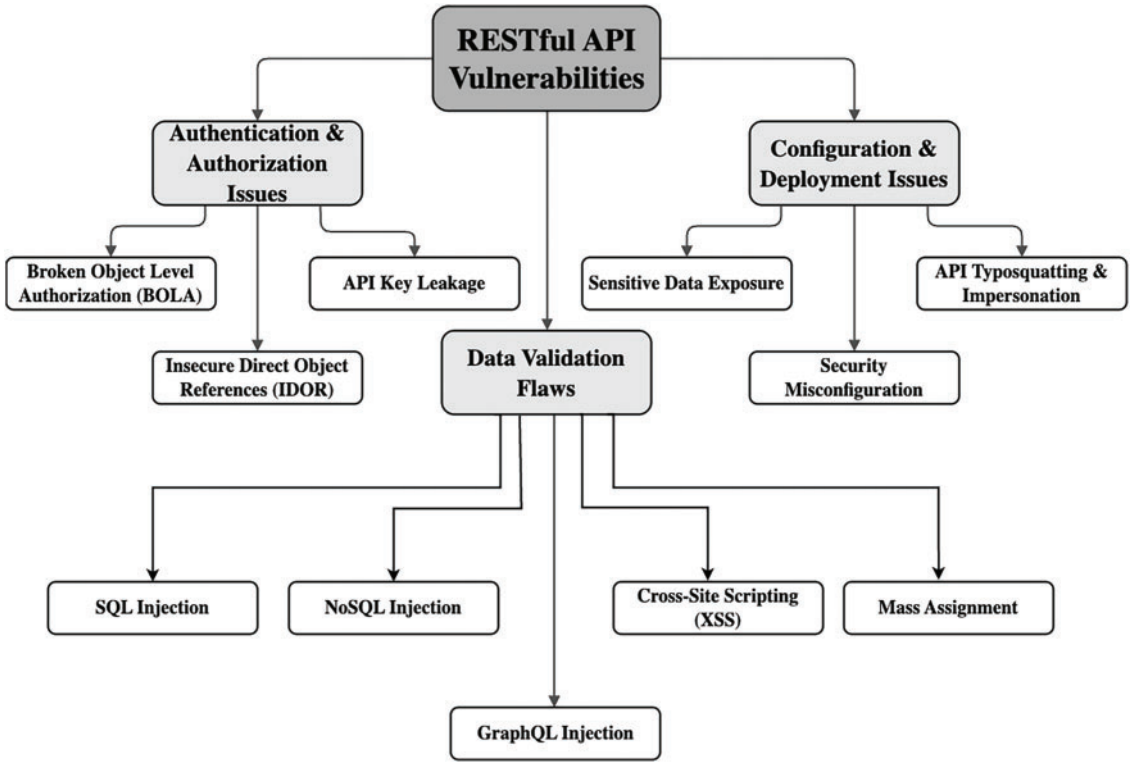

Figure 6: Taxonomy of RESTful API vulnerabilities

4.1 Authentication and Authorization

Insecure Direct Object References (IDOR)

IDOR is an object reference vulnerability where an API directly maps an internal implementation object like a file or a database key to the user. This type of vulnerability is prevalent in APIs that fail to enforce strict access controls and is often found in URLs, request headers, or payloads where sensitive data identifiers are directly exposed. This can be exploited by the attackers by manipulating these references so as to access resources that they are not supposed to. For example, if a URL contains the user ID or file ID, then the attacker can change the ID value to gain access to other user's data or restricted files.

Impact: IDOR vulnerabilities can result in data leakage, alterations to data, and even potential data breaches. They are especially risky due to the fact that they may be utilized in a malicious manner without any special equipment or understanding. The impact can range from minor data leakage to significant privacy violations, depending on the nature of the exposed data [12].

Exploitation Example: An attacker navigates a URL like

https://api.example.com/user/profile?id=123 (accessed on 23 June 2025)

and changes id = 123 to id = 124, thereby accessing another user's profile without proper authorization checks. This manipulation exploits the lack of access control checks tied to each object's ID, allowing unauthorized data access.

Detection Methods: IDOR vulnerabilities can be detected through a combination of automated tools and manual testing. Tools and techniques used often involve analyzing URL patterns and identifying instances where sensitive data is exposed in the URL. Automated tools can analyze API endpoints to identify

patterns where sensitive data is exposed, simulating various attack scenarios by altering request parameters to check for unauthorized access. Manual penetration testing complements this by exploring non-standard input methods that automated tools might overlook [12].

Broken Object Level Authorization (BOLA)

Broken Object Level Authorization (BOLA) arises when an API lacks correct authorization checks at the object level. This may lead to privilege escalation whereby a user with restricted access is able to access other restricted areas or perform operations that should only be done by users with higher privileges. BOLA vulnerabilities are prevalent in the APIs that only use user roles without checking object-specific permissions. BOLA vulnerabilities occur when APIs rely solely on user roles and fail to verify permissions for individual objects, allowing users to access or modify resources beyond their intended permissions [12].

Impact: BOLA vulnerabilities compromise data integrity and confidentiality by enabling unauthorized data access and manipulation. This can lead to privilege escalation attacks, where attackers exploit insufficient authorization checks to perform administrative actions or access sensitive information. The consequences include data breaches, loss of trust, and potential financial and legal repercussions [12].

Exploitation Example: An API allows users to view documents based on their roles. A regular user modifies the request to include document IDs assigned to administrators, thereby accessing confidential documents without proper authorization checks. This highlights the API's failure to enforce object-level permissions, allowing unauthorized access.

Detection Methods: Identifying BOLA vulnerabilities calls for a thorough review of API access controls to ensure object-level checks are implemented. Automated tools simulate various user roles, attempting to access or modify objects beyond their privilege levels. Effective detection involves continuous validation of permissions, particularly as APIs evolve or roles change. Implementing least privilege principles and conducting regular audits of access controls can help mitigate BOLA vulnerabilities [12].

Distinction between IDOR and BOLA

While both Insecure Direct Object Reference (IDOR) and Broken Object Level Authorization (BOLA) involve unauthorized access to resources, they differ in technical manifestation and exploitation. IDOR typically refers to cases where user-controllable identifiers (such as IDs in URLs) are insufficiently protected, enabling attackers to access or manipulate objects by guessing or enumerating identifiers. BOLA, as defined in the OWASP API Security Top 10 [9], is a broader category that encompasses any failures in enforcing object-level authorization, which may or may not involve direct references. In practice, all IDORs are instances of BOLA, but BOLA also covers authorization logic flaws not strictly related to object references. We separate these categories to provide fine-grained guidance for detection and remediation.

API Key Leakage

API key leakage is a significant vulnerability where API keys are inadvertently exposed in public repositories, logs, or client-side code. Attackers can exploit these leaked keys to gain unauthorized access to APIs, leading to resource exhaustion, data theft, and API abuse.

Impact: Leaked API keys allow adversaries to execute API requests with full privileges, resulting in unauthorized data access, account takeover, and service disruption. Studies analyzing the RapidAPI ecosystem found that over 3500 API keys were publicly exposed on GitHub and mobile applications, with 98% of them still active [22].

Exploitation Example: An attacker finds an API key hardcoded in a mobile application's source code. Using this key, they gain unauthorized access to premium API services, leading to Theft of Service attacks, as observed in the RapidAPI security study [22].

Detection Methods: API security tools such as GitGuardian and TruffleHog can scan repositories for exposed API keys. Additionally, runtime monitoring tools can track abnormal API usage to detect compromised keys. Implementing automated key rotation and scope-restricted API keys mitigates this risk [22].

4.2 Data Validation Flaws

SQL Injection

SQL Injection vulnerabilities occur when an API allows unsanitized user input to be included in SQL queries. Attackers exploit this by injecting malicious SQL code into input fields, altering database operations to retrieve or manipulate sensitive data. This vulnerability arises when user inputs are directly included in SQL statements without adequate validation or parameterization, allowing attackers to execute arbitrary SQL commands.

Impact: SQL Injection can have devastating effects, including unauthorized data access, data corruption, and complete database compromise. Attackers can extract sensitive information, modify or delete data, and even gain administrative control over the database. SQL Injection has the potential to disrupt business operations, compromise user privacy, and lead to financial losses [13,30].

Exploitation Example: An attacker inputs ‘ OR ’1‘=’1 into a login form, causing the SQL query to always evaluate to true and granting unauthorized access to user accounts. This technique bypasses authentication mechanisms, exposing user data and potentially allowing the attacker to manipulate the database further.

Detection Methods: Detecting SQL Injection involves using automated tools that employ input fuzzing and pattern recognition to identify vulnerable query structures. These tools simulate various payloads mimicking common SQL injection techniques to determine if the application improperly executes them. Prevention strategies include using parameterized queries, prepared statements, and stored procedures to separate user inputs from SQL code. Regular code reviews and vulnerability assessments are essential to maintain secure coding practices [1,11].

NoSQL Injection

NoSQL Injection vulnerabilities occur when APIs incorporate unsanitized user input into NoSQL queries. Attackers exploit this to alter query logic and access or manipulate data within NoSQL databases, which are commonly used for their flexibility and scalability. Unlike SQL databases, NoSQL databases often lack strict schema enforcement, making them susceptible to injection attacks.

Impact: NoSQL Injection can lead to unauthorized data access, data manipulation, and potential compromise of entire databases. These vulnerabilities pose significant risks as NoSQL databases are increasingly used in web applications, often holding large volumes of unstructured data. The impact includes data breaches, loss of data integrity, and exposure of sensitive information [1].

Exploitation Example: An attacker crafts a request that modifies a NoSQL query structure, such as injecting additional fields or altering query conditions, to bypass access controls and retrieve unauthorized data. This exploitation highlights the lack of input validation and query sanitization in NoSQL implementations.

Detection Methods: Detection tools for NoSQL Injection utilize input validation checks and query sanitization techniques. They simulate various inputs to ensure queries maintain their intended logic and do not execute arbitrary commands based on untrusted data. Regular audits, the use of security libraries that enforce strict input validation, and implementing whitelisting of expected inputs can help prevent NoSQL Injection [1].

GraphQL Injection

GraphQL APIs introduce new attack vectors due to their flexible query structures. Unlike REST APIs, where endpoints define data retrieval, GraphQL allows clients to structure queries dynamically, increasing the risk of over-fetching, denial-of-service (DoS), and injection attacks.

Impact: GraphQL Injection enables attackers to modify query structures, bypass authorization, and exfiltrate sensitive data. It can lead to excessive server load, data exposure, and unauthorized operations [31].

Exploitation Example: An attacker modifies a GraphQL query to request fields beyond their authorization scope:

```
 {
user(id: "123")  {
  id
  passwordHash
  creditCardInfo
 }
}
```

This allows unauthorized access to private user information, leading to data breaches.

Detection Methods: Detection strategies for GraphQL Injection include query complexity analysis, introspection restrictions, and rate limiting. Security tools like GraphQL Armor and GraphQL Security Scanner can help mitigate these risks [31].

Command Injection

Command Injection vulnerabilities occur when unsanitized user inputs are included in system commands executed by the application. Attackers exploit this by injecting malicious commands, potentially executing arbitrary operations on the host system, leading to unauthorized access or control. This vulnerability arises when user inputs are directly passed to system command functions without proper validation.

Impact: Command Injection can severely compromise system security, allowing attackers to execute arbitrary commands, access sensitive files, and manipulate system configurations. This makes it a critical security vulnerability with the potential for significant damage, including data loss, service disruption, and unauthorized access to underlying infrastructure [2].

Exploitation Example: An attacker submits a command like ; rm -rf/in an input field that gets executed on the server, resulting in critical system damage or data loss. This example demonstrates how unvalidated inputs can be leveraged to perform destructive actions.

Detection Methods: Detecting Command Injection involves input validation and command filtering. Security tools inspect user inputs and command constructs to identify and block potential injection attempts before execution. Prevention strategies include using parameterized command execution, avoiding the use of shell commands for processing user inputs, and employing security libraries that validate and sanitize inputs [2].

Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) vulnerabilities occur when an API allows the injection of malicious scripts into web pages viewed by other users. These scripts execute in the context of the victim's browser, potentially stealing cookies, session tokens, or executing actions on behalf of the user. XSS vulnerabilities arise when user inputs are reflected in web pages without proper validation and sanitization.

Impact: XSS attacks compromise user data, deface websites, and propagate malware, affecting the confidentiality and integrity of user interactions with web applications. They can result in the theft of sensitive information, unauthorized actions, and the spread of malicious content, posing a significant threat to user privacy and application security [10,32].

Exploitation Example: An attacker injects a script in a comment section that executes when other users view the page, stealing their session cookies and allowing the attacker to impersonate them. This exploitation illustrates the potential for XSS to facilitate identity theft and unauthorized actions.

Detection Methods: XSS detection tools analyze API responses and web application behavior to identify improper input handling and script execution. Automated tools simulate various attack vectors to uncover vulnerabilities, ensuring that user inputs are sanitized and not executed as code. Prevention measures include implementing strict input validation, output encoding, and using security libraries that automatically sanitize user inputs [10,13,32].

Justification for Inclusion: XSS vulnerabilities can have a significant impact on REST APIs, particularly when these APIs handle user-generated content or are integrated with web applications. A study by Zhang et al. conducted a systematic analysis on XSS attacks in RESTful APIs, highlighting that APIs are increasingly targeted due to their role in rendering and exchanging user content within web applications. The research found that APIs, even when not directly exposed to the web, can still propagate XSS vulnerabilities when connected to web interfaces, making them a critical consideration in API security [33].

Furthermore, a recent advisory documented a stored XSS vulnerability in a REST API, where a malicious script could be executed in the browser of an authenticated user viewing data through the API, leading to potential compromise of the user's session and unauthorized actions [34].

Mass Assignment Vulnerabilities:

Mass assignment vulnerabilities occur when an API allows clients to update object fields without explicit permission checks. This happens when user inputs are automatically mapped to data models without verifying which fields should be modifiable. Attackers exploit this by including unauthorized fields in their requests, leading to unintended modifications of sensitive data like user roles or permissions.

Impact: The impact of mass assignment vulnerabilities is significant, as they can lead to unauthorized data manipulation and compromise the integrity of the system. Attackers might change critical fields, such as user roles, permissions, or configuration settings, potentially resulting in privilege escalation. Such vulnerabilities can also lead to data corruption, unauthorized transactions, and financial loss [16].

Exploitation Example: In an e-commerce API that allows users to update profile information, if the API doesn't validate which fields are updatable, an attacker could include isAdmin=true in their request to elevate their privileges. This occurs because the system improperly trusts user-provided data and updates object properties indiscriminately.

Detection Methods: Static analysis is used which examines the code and specifications without its execution. This allows for the early detection of vulnerabilities, helping prevent potential exploits during deployment. By scrutinizing API documentation and specifications, static analysis identifies fields that may be vulnerable to unauthorized modification, enabling developers to implement necessary safeguards before deployment [16].

4.3 Configuration and Deployment Issues

Misconfiguration

Misconfiguration vulnerabilities occur when APIs are deployed with insecure default settings or improperly configured security controls. Common misconfigurations include exposed debugging information, overly permissive CORS (Cross-Origin Resource Sharing) settings, and lack of encryption for sensitive data. Such vulnerabilities often result from oversight or inadequate configuration management [13].

Impact: Misconfigurations can lead to unauthorized access, information leakage, and exploitation of other vulnerabilities. They are often easily exploited by attackers due to inherent weaknesses in configuration settings, potentially leading to data breaches and system compromise. The consequences of misconfiguration are particularly severe as they can expose entire systems to unauthorized access and attacks.

Exploitation Example: An API with debug mode enabled exposes detailed server error messages, providing attackers with information to craft more targeted attacks or exploit other vulnerabilities. This example highlights how seemingly minor misconfigurations can have significant security implications.

Detection Methods: Detecting misconfigurations involves comprehensive security audits and configuration reviews. Tools compare current configurations against security best practices and guidelines to identify weaknesses, ensuring APIs are securely configured and maintained. Regular monitoring, automated configuration management tools, and adherence to security benchmarks can help mitigate misconfiguration risks [13].

Typosquatting and API Impersonation

Recent studies reveal that malicious actors can publish fraudulent APIs that mimic legitimate services. This attack, known as API Typosquatting, targets developers who mistakenly integrate similarly named APIs into their applications.

Impact: Attackers use typosquatting to impersonate well-known APIs, tricking developers into integrating malicious APIs that steal data, inject malware, or degrade application functionality. Research into the RapidAPI platform found multiple instances of malicious APIs impersonating reputable services to deceive developers [22].

Exploitation Example: A developer intending to integrate the Stripe API mistakenly uses a malicious API named Str1pe hosted on an API marketplace. This fraudulent API captures payment credentials, resulting in financial fraud.

Detection Methods: Developers should verify API sources, cross-check API metadata, and use code-signing mechanisms to authenticate API providers. Marketplace platforms should enforce stricter verification procedures to prevent fraudulent API registrations [22].

Exposure of Sensitive Information

Exposure of Sensitive Information occurs when APIs inadvertently disclose sensitive data through responses or error messages. This can include information such as database connection strings, API keys, or user credentials, often exposed due to insufficient error handling or logging practices.

Impact: Exposing sensitive information can lead to severe security breaches, unauthorized access, and data leaks. It undermines API confidentiality and integrity, providing attackers with critical details to further exploit the system. The impact of such exposures can be far-reaching, potentially leading to identity theft, financial loss, and reputational damage for affected organizations.

Exploitation Example: Error messages reveal stack traces containing sensitive database connection details, which attackers use to access and manipulate the database or other systems. This exploitation demonstrates how improper error handling can inadvertently disclose critical information.

Detection Methods: Tools designed to detect sensitive information exposure analyze API responses and logs for patterns that match sensitive data, ensuring no critical information is inadvertently disclosed. Implementing proper error handling and logging practices, along with regular audits, can prevent such exposures. Encrypting sensitive information in transit and at rest, and restricting access to logs and error messages, further enhances security [35].

4.4 Comparison with Existing Taxonomies

While the OWASP API Security Top 10 [9] and industry reports like Salt Security and Traceable AI provide high-level classifications of API threats, they do not offer a hierarchical or empirically validated taxonomy specific to RESTful APIs. Our taxonomy differs by introducing:

- **Hierarchical Structure:** Groups threats by impact, exploitation vector, and detection complexity.
- **RESTful-Specific Categories:** Includes API-exclusive risks such as Mass Assignment and Automated Scraping, absent from traditional web security taxonomies.
- **Empirical Validation:** Integrates recent real-world breach data and academic findings to ensure practical relevance.

Table 3 provides a comparative overview of major API vulnerability taxonomies, highlighting the unique hierarchical and RESTful-specific nature of our proposed approach.

Table 3: Comparison of API vulnerability taxonomies

Source	Classification approach	RESTful-specific?
OWASP API Top 10 (2023)	High-level risk categories (non-hierarchical)	No (General API risks)
Salt security report	Industry trends, attack examples	Partially (No formal taxonomy)
This work (Proposed)	Hierarchical, RESTful-specific, empirically validated	**Yes** (e.g., Mass assignment, Automated scraping)

5 Sanitization Methods and Routine Functions

The security of web APIs relies significantly on strong input sanitization methods that help prevent common attacks like Cross-Site Scripting (XSS), SQL Injection (SQLi), and Regular Expression Denial of Service (ReDoS). Without proper sanitization, attackers can take advantage of API endpoints, resulting in data breaches and service interruptions. This section describes the basic sanitization routines used to protect API requests, emphasizing input validation, escaping, filtering, and encoding techniques.

5.1 Input Validation and Filtering

Input validation ensures that only expected and well-formed data is processed by the API. Several studies highlight that improper validation can expose services to ReDoS attacks, where inefficient regular expressions lead to computational exhaustion [19].

5.1.1 Regular Expression-Based Input Sanitization

Web services commonly use regular expressions (regexes) to validate inputs in HTML forms and OpenAPI specifications. However, regexes that allow unbounded backtracking can introduce ReDoS vulnerabilities [19].

Identified Risks:

Some services disclose client-side regex sanitization patterns, which attackers can use to craft slow-matching regex attacks [19]. Weak regex patterns allow adversaries to bypass validation and inject malicious payloads.

Mitigation Strategies:

- Limit input length for regex-based validation.
- Implement timeout mechanisms for regex evaluation.
- Use linear-time regex engines to prevent backtracking.

5.1.2 Static and Dynamic Analysis-Based Sanitization

Automated approaches for sanitization synthesis leverage static analysis to detect unvalidated inputs and dynamically filter malicious data before it reaches security-sensitive functions [18].

- **Static Analysis:** Uses automata-based techniques to identify insecure API inputs.
- **Dynamic Analysis:** Detects and blocks malicious payloads at runtime.

5.2 Encoding and Escaping Techniques

Encoding and escaping prevent attacks by transforming special characters into safe representations before processing. These techniques are commonly used to mitigate XSS and injection attacks [36].

5.2.1 Escaping Special Characters

- **HTML & JavaScript Escaping:** Converts special characters into safe entities to prevent script execution.

```
htmlspecialchars() (PHP)
encodeURIComponent() (JavaScript)
json.dumps(input, ensure_ascii=True) (Python)
```

- **SQL Query Escaping:** Prevents SQL Injection by escaping special symbols.

```
mysqli_real_escape_string() in PHP
```

5.2.2 Content Security Policy (CSP) Enforcement

Defines trusted sources for content execution to prevent malicious scripts from loading.

5.3 SQL Injection Prevention via Parameterized Queries

SQL Injection remains one of the most critical API security risks, allowing adversaries to execute arbitrary database queries [17]. Sanitization routines that rely on escaping are often insufficient, and prepared statements provide stronger protection [17].

Best Practices:

- Always use parameterized queries instead of string concatenation.
- Avoid dynamic query construction with unsanitized user input.

5.4 Automated Sanitization Patch Generation

Recent studies propose automated sanitization patch generation, which synthesizes sanitization routines dynamically based on observed API threats [18].

- **Match-and-Block Strategy:** Detects malicious input and blocks execution.

- **Match-and-Sanitize Strategy:** Identifies malicious input and modifies it minimally to prevent attacks.

5.5 Mitigation Strategies for Key RESTful API Vulnerabilities

Table 4 summarizes recommended mitigation techniques for the primary classes of vulnerabilities identified in our taxonomy. Each mitigation strategy is supported by recent studies and established best practices from peer-reviewed literature.

Table 4: Mitigation strategies for RESTful API vulnerabilities

Vulnerability	Threat description	Mitigation strategies	Supporting literature
IDOR/BOLA	Unauthorized access to resources by manipulating object references or lack of object-level permission checks.	Enforce strict object-level authorization for every operation; avoid predictable IDs; use indirect references; employ logic testing (e.g., RESTlogic).	[14]
Injection (SQLi, NoSQLi)	Malicious code execution in backend databases via unsanitized inputs.	Use parameterized queries, rigorous input validation, ORM libraries, restrict DB permissions; fuzzing for injection flaws.	[14,37]
Broken authentication	Attackers gain unauthorized access due to weak auth schemes or poor session management.	Use OAuth 2.0/OpenID Connect, enable MFA, set proper session timeouts, promptly invalidate tokens, strong password policies.	[14,38]
Excessive data exposure	Overly permissive APIs return more data than needed, risking data leaks.	Output filtering, whitelist response fields, validate schemas, never send sensitive data by default, minimize data in responses.	[39]
Security misconfiguration	Weak default settings, unpatched services, excessive permissions, misconfigured endpoints.	Automate config management, apply least privilege, regular updates/patches, CI/CD pipelines for security, periodic audits.	[40]

By embedding these detailed mitigation strategies and linking explicitly to prior sections and literature, the comprehensiveness and academic rigor of this subsection are significantly enhanced.

6 Gaps in Current Research

Based on our systematic review and analysis of the current literature, we have outlined several important research limitations regarding RESTful API vulnerability detection. It is imperative to fill these gaps to progress the field and improve the effectiveness and reliability of security.

6.1 Limited Real-World Application

A significant number of existing tools and methodologies on RESTful API vulnerability detection are either tested using synthetic benchmarks or in controlled settings. Despite the fact that such controlled environments are useful for research and facilitate hypothesis testing, they do not necessarily reflect real-world situations and conditions [11].

Definition of Real-World Applications: Real-world applications refer to software systems that are actively deployed and used in practical, everyday scenarios outside of controlled or simulated environments. These applications are complex, scale to handle real-world data, and are subject to various operational constraints such as performance, security, and integration with other systems.

Challenge: The difference between laboratory conditions and real-world applications implies that a tool may work well in test conditions but may not run as smoothly in real-life application. Real-world APIs exhibit various complexities in terms of configuration, usage, and variability that cannot be easily emulated within an artificial context [32].

Significance of Real-World Applications: Validating tools against real-world applications ensures that they can handle the complexity, scale, and unpredictability of live environments. This approach exposes potential vulnerabilities that may not manifest in test environments, making the tools more robust and effective in detecting and mitigating vulnerabilities under real-world conditions.

Need: More studies are required that aim at proving the efficiency of the given vulnerability detection tools on real-world applications and datasets. This means working with industry partners to obtain access to a variety of API implementations that are representative of those found in practice. Research should also compare the effectiveness of these tools in real-life situations, taking into account the load of the API, the time to complete responses, and the ability to interface with other services.

6.2 Incomplete Dependency Detection

Dependency detection between API operations has been identified to be one of the biggest problems in RESTful API vulnerability detection. Dependencies can exist when one API operation depends on the result of another operation or when several operations are to be performed in order to be effective [11].

Challenge: Current tools often fail to recognize and address these dependencies effectively, which results in incomplete testing and potential vulnerabilities. Dependencies can also be intricate and may depend on the environment; thus, they are challenging to identify using static analysis only [32].

Why Higher-Level Algorithms and Machine Learning Techniques Are Needed: Detecting these dependencies is challenging due to the complex interactions, dynamic nature, and hidden states within modern API ecosystems. Simple heuristic-based methods often fall short in capturing these complexities.

1. Complex Interactions: Modern distributed systems involve multiple services, databases, and external systems, creating intricate dependencies that are difficult to trace.
2. Dynamic Nature of APIs: APIs are constantly evolving, making it difficult to maintain an up-to-date understanding of dependencies.
3. Indirect Dependencies and Hidden States: Some dependencies are not explicitly documented but are embedded within the application logic.

Need: Dependency detection requires higher-level algorithms and machine learning techniques. These techniques should be able to parse sequences of API sequence calls, comprehend the dependencies between the various operations, and discover emerging security threats that originate from these dependencies.

Advanced Techniques for Improved Dependency Detection:

1. Machine Learning Models: Sequence-to-sequence models, recurrent neural networks (RNNs), and reinforcement learning can analyze API interaction data to predict and detect hidden dependencies. Early deep-learning classifiers leveraging character n-gram embeddings [41] and kernel-level eBPF-based observability frameworks [42] have shown promise in automated vulnerability detection.
2. Graph-Based Models: Dependency graphs and algorithms like depth-first search can manage and visualize both direct and indirect dependencies, providing a comprehensive view of potential vulnerabilities.

6.3 Dynamic and Evolving APIs

APIs are dynamic and often undergo changes, and this means that there could be a difference between what the API specification states and the actual implementation. Such changes can create new risks or influence the efficiency of the existing security measures [30].

Challenge: The constant development of APIs is a problem for automated vulnerability detection tools. Tools need to track changes in API endpoints, parameters and data flows which may change over time [10].

Need: Automated tools that can continuously validate and reconcile discrepancies between API specifications and implementations are essential. These tools should be able to track API changes in real-time, adapt the detection models to these changes, and guarantee that new added features or changes to existing ones do not introduce security vulnerabilities.

6.4 Comprehensive Specification Analysis

Given the fact that RESTful APIs are widely used and continuously developed, the analysis of API specifications is an important step to guarantee the security and stability of software systems. Specifications give a clear description of how the API is expected to function, the exact endpoints, parameters, data type and the access control measures [10].

Challenge: Currently, there are few if any tools that can parse API specifications and analyze them to find errors, omissions, or inaccuracies. When specifications are not clearly defined or are incorrect, the result is gaps in security that cannot be identified merely by examining the code [11].

Need: There is a need for tools that are capable of processing the API specifications and then checking the actual implementation against it. These tools must be capable of pointing out gaps, omissions and errors and provide suggestions on how to rectify these flaws so that specified procedures conform to the API's functionality.

As illustrated in Fig. 7, our work builds upon previous studies and identifies key gaps in API security literature, paving the way for structured future research.

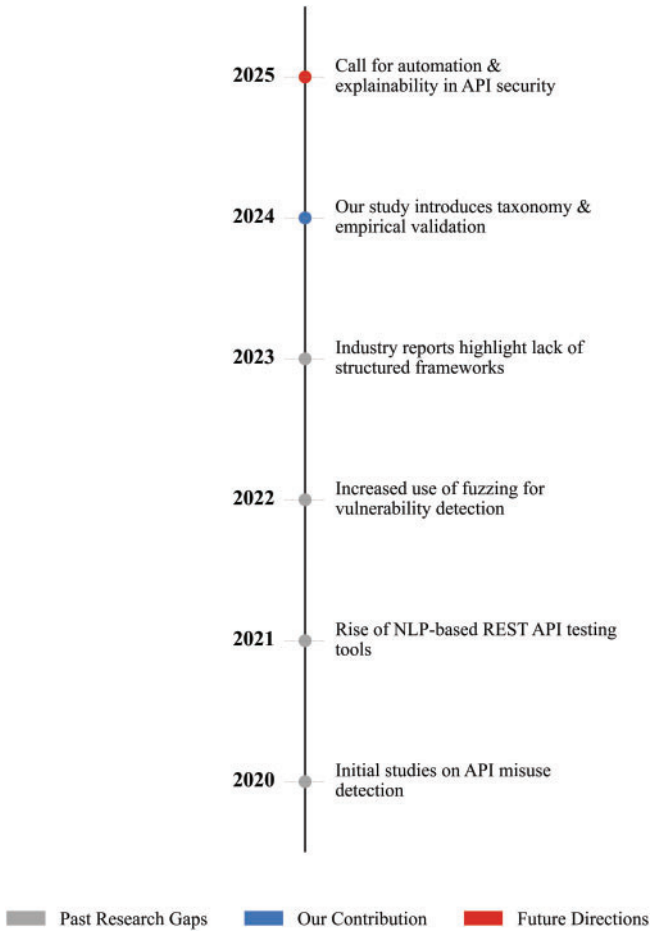

Figure 7: A timeline overview of major research gaps, our contributions, and future directions in **RESTful** API security research

7 Empirical Validation

The detection of RESTful API vulnerability has evolved for the past years to show better accuracy and coverage by using several techniques.

Real-World Validation: Highlighted Case Studies

Our empirical validation draws upon multiple real-world case studies, each illustrating distinct classes of RESTful API vulnerabilities:

- **SQL Injection in Healthcare and Banking APIs:** Integration of API firewalls and CSRF token-based detection prevented data breaches in production systems.
- **RESTlogic in Cloud Services:** Discovery of a previously unknown privilege escalation vulnerability in OpenStack, validating the practical effectiveness of logic testing tools.
- **Bitbucket API Testing:** Identification of a chain of API operations triggering critical internal errors, demonstrating the value of adaptive sequence testing.
- **Access Control Flaws in Financial APIs:** Automated OpenAPI specification analysis exposed hidden IDOR vulnerabilities, directly preventing unauthorized data access.

These cases demonstrate that theoretical findings translate into measurable improvements in real-world API security. Practical resources such as Vulnerability Lab's technical attack sheet [43] further underscore the variety of real-world vectors our taxonomy must address.

Fig. 8 shows a comparison between false positive and false negative cases for different detection methods and their performances as discussed in [12]. This comparison highlights the need to strike a right balance between detection accuracy and false positives and negatives which is very important when deploying these tools in real life situations.

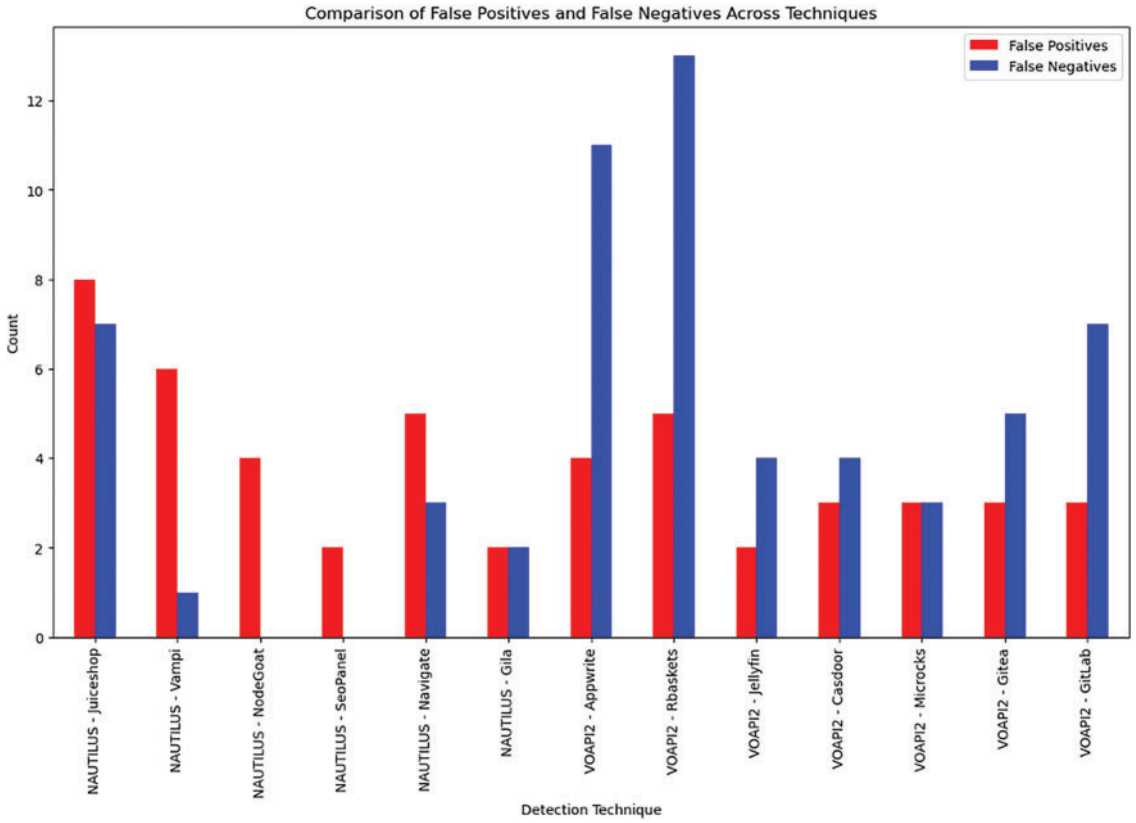

Figure 8: Comparison of false positives and false negatives across different RESTful API vulnerability detection techniques. Adapted from [13]

Table 5 presents precision, recall, and F1-score comparisons for various API security detection models. RESTlogic outperforms other methods, particularly in reducing false positives. Morest provides strong adaptive testing but with slightly higher false positives, while the SQL Injection API model remains effective in targeted injection detection.

Table 5: Performance metrics of API vulnerability detection methods. Adapted from [1,2,14,30,32]

Method	Precision (%)	Recall (%)	F1-Score (%)	False Positives (%)
RESTlogic (Call stack anomaly detection)	94.2	91.8	93.0	3.5
Morest (Adaptive API sequence testing)	90.1	87.5	88.8	5.2

(Continued)

Table 5 (continued)

Method	Precision (%)	Recall (%)	F1-Score (%)	False Positives (%)
Bitbucket API testing (Rule-Based)	85.4	80.9	83.1	6.7
SQL Injection API model (WAF+CSRF Token)	89.9	86.2	88.0	4.9

This section highlights more key empirical validations from recent research.

1. **Case Study: Detection of SQL Injection Web API**
   Background: Multiple web applications including those in the healthcare and banking sectors were found to be facing frequent security breaches, leading to unauthorized access to sensitive data. The application lacked adequate security measures to prevent common web vulnerabilities such as SQL injection and Cross-Site Scripting (XSS) [1,2].
   Implementation: The proposed RESTful API-based vulnerability detection model was integrated into the application. The model employed a Web Application Firewall (WAF) to monitor and filter incoming requests for potential vulnerabilities. The API authenticated requests using API keys stored in an encrypted format and validated requests using CSRF tokens for POST and GET methods [1].
   Results: The integration of the model significantly reduced the number of successful attacks. The model detected and blocked multiple SQL injection attempts and XSS attacks, preventing unauthorized access to sensitive data. Web apps now experienced a marked decrease in security breaches within the first three months of implementation [2].
2. **Case Study: RESTlogic in Cloud REST APIs**
   Background: The effectiveness of RESTlogic was ascertained through experiments on real-world open-source cloud services such as OpenStack. These experiments revealed several cases of API noncompliance with their specifications, confirmed previously reported vulnerabilities, and revealed previously unknown logical vulnerabilities.
   Implementation: Integrating parameter inference and call stack anomaly detection of RESTlogic provided much better coverage and depth in API logic testing.
   Results: A previously undiscovered logical flaw in OpenStack's resource management API was discovered, demonstrating the functional application of RESTlogic [14].
3. **Case Study: Bitbucket**
   Background: Bitbucket, a Git-based source code repository hosting service owned by Atlassian, was used as a subject in evaluating the Morest model-based RESTful API testing technique. During the testing, specific call sequences that triggered internal server errors were identified.
   Implementation: The testing process involved creating a project through the /rest/.../projects endpoint via a POST operation. The project information could be further retrieved by a GET request on the same endpoint. The next step was to create a repository in the project via the /rest/.../projectKey/repos endpoint through a POST operation, where the projectKey parameter was defined in the first step as a parameter. Finally, a GET query on /rest/.../repos/repositorySlug/commits with parameters

   ```
   {"path": "test_string"}
   ```

   triggered an internal server error.
   Results: When debugging mode was enabled, the bug message was printed out:

   ```
   "com.atlassian.bitbucket.scm.CommandFailedException"
   ```

- This sequence demonstrated that both a project and a repository needed to be created first to trigger this bug. With RPG guidance and dynamic RPG updating, Morest adaptively generated such call sequences [32].

4. **Case Study: Detection of Access Control Vulnerabilities Background:** The algorithm proposed for detecting IDOR and BOLA vulnerabilities through OpenAPI specification analysis was demonstrated and tested on a number of microservice architectures.
   Implementation: The findings revealed that the algorithm was useful in detecting access control issues that may compromise the data's security. For instance, identification of an IDOR vulnerability in a financial service API helped avoid data breaches by denying unnecessary access to sensitive resources.
   Results: The implementation successfully detected several access control vulnerabilities, enhancing the security of the tested microservice architectures [12].

Table 6 compares API vulnerability detection methods. RESTlogic is highly precise with low false positives, making it suitable for cloud-based REST API security. Morest provides strong adaptation in API misuse scenarios but has higher false positives. Bitbucket API testing is effective for repository-based vulnerabilities, while SQL Injection models provide structured attack prevention.

Table 6: Comparison of vulnerability detection methods. Adapted from [1,2,14,30,32]

Detection method	Strengths	Weaknesses
RESTlogic	High detection accuracy, call stack anomaly detection, low false positives	Requires deep API behavioral modeling
Morest	Effective for adaptive API sequence testing, discovers unknown vulnerabilities	Slightly higher false positives due to dynamic updating
Bitbucket API testing	Finds internal server errors, useful for repository APIs	Limited to predefined patterns
SQL injection API model	Detects SQL-based injection attacks effectively, robust against tampering	Requires manual rule configuration

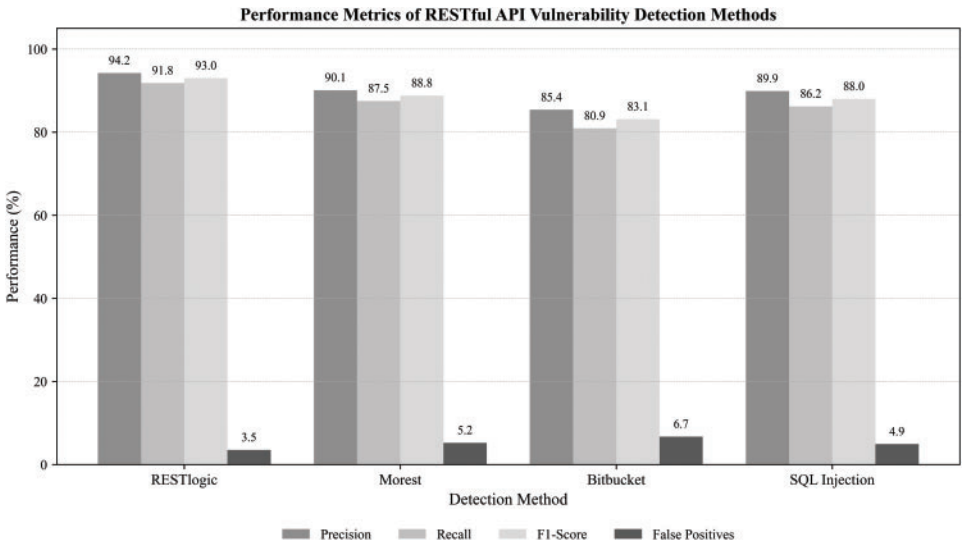

Figure 9: Comparative performance metrics (Precision, Recall, F1-Score, and False Positives) of leading RESTful API vulnerability detection models

As shown in Fig. 9, RESTlogic significantly outperforms other models in precision and false positive rate, indicating its practical advantage for robust deployment.

8 Discussion and Practical Implications

Emerging API paradigms such as GraphQL introduce unique vulnerabilities [44,45], while few-shot classification defenses [46] and intelligent fuzzers like LlamaRestTest [47] point toward automated anomaly detection. Explainable AI approaches also offer insight into attack patterns [48].

8.1 Cost-Benefit Considerations

Integrating robust security mechanisms into API development processes is essential yet incurs significant economic implications. Organizations must judiciously evaluate the balance between the advantages of sophisticated detection methodologies, such as automated fuzzing, dynamic analysis, and formal specification verifications, against their associated costs, required expertise, and operational overhead. Advanced approaches, while significantly enhancing vulnerability coverage and reducing long-term security incidents, demand considerable initial investment in tooling, training, and continuous maintenance [14,40].

Studies highlight that incorporating security controls early in software development significantly reduces remediation expenses, potentially up to 100 times less costly compared to addressing vulnerabilities post-deployment [40]. The 2025 API Security Impact Study corroborates this, reporting average API-related incident costs exceeding half a million dollars annually, emphasizing the financial rationale for preemptive investment in API security [49]. Moreover, economic modeling by Kong forecasts a global economic impact attributed to APIs reaching $17.3 trillion by 2030, underlining the strategic importance of securing these infrastructures against escalating AI-enhanced threats projected to increase by 548% by 2030 [38].

Decision-makers must thus consider their organization's threat landscape, compliance obligations, and resource constraints to deploy the most economically viable security strategy. A strategic focus on automated input generation via natural language processing (NLP) and machine learning (ML), along with real-time monitoring tools, can increase testing coverage and effectiveness, thereby optimizing return on security investment [14].

8.2 Impact on Usability and Developer Experience

The integration of comprehensive security protocols within API design, while indispensable, directly influences API usability and developer productivity. Excessive security measures such as stringent access controls, complicated authentication procedures, or cumbersome workflows can obstruct seamless API integration and diminish developer efficiency and adoption rates [14,39].

Effective API usability is achieved by adhering to clearly defined RESTful design principles, employing consistent naming conventions, comprehensive documentation, and intuitive endpoint designs [39,50]. Multiple works assert that APIs adhering to consistent naming conventions and standardized data formats like JSON enhance developer adoption significantly, by up to 37% and 28%, respectively, underscoring the direct relationship between thoughtful API design and improved developer experience [39].

Additionally, overly complex or opaque security configurations can lead to increased cognitive load for developers, reducing efficiency and productivity. Usability is further compromised when API specifications are misaligned with actual implementations, a frequent occurrence highlighted in recent systematic reviews [50]. Thus, aligning API documentation and implementation accurately through continuous specification validation, such as through automated CI/CD practices, is crucial to enhance developer experience [14].

Striking an optimal balance necessitates employing 'secure-by-design' strategies, integrating security seamlessly within development lifecycles, and leveraging user-friendly authentication mechanisms (e.g., OAuth 2.0). This approach ensures APIs are not only robustly secure but also developer-friendly, fostering productive collaboration between security and development teams and enhancing overall API adoption and productivity [14,38].

9 Future Research Directions

To further develop the area of RESTful API vulnerability detection, the following directions should be considered. These directions aim at mitigating current limitation and improving the reliability and efficiency of the approaches for vulnerability identification.

9.1 Enhanced Input Generation

Input generation is one of the most important steps of vulnerability detection because it defines the range and the variety of test cases to be used to attack an API. Future work should focus on enhancing the input synthesis strategies to address the issue of domain-specific constraints and dynamic values.

Leveraging Large Language Models (LLMs): Recent advancements, such as RESTGPT by Kim et al., highlight the potential of Large Language Models to extract machine-interpretable rules from natural-language descriptions within API specifications. By using LLMs to improve context-awareness and accuracy in input generation, researchers can better identify complex, context-dependent vulnerabilities. This approach can greatly enhance the precision of REST API testing tools, making them more effective in real-world scenarios [51]. Combining reinforcement-learning fuzzers like WENDIGO [52] with formal OpenAPI-to-Petri-net transforms [53] represents a promising path to fully automated, stateful vulnerability discovery.

Leveraging NLP: Natural Language Processing (NLP) can be utilized to extract meaningful values from API specifications, server logs, and documentation. NLP can also create realistic and context-sensitive input values based on textual descriptions and parameters that are likely to reveal vulnerabilities. For example, extracting common user inputs and restrictions from API documentation can be useful in developing test scenarios that are similar to real-world usage cases [54,55].

NLP-Driven Specification Enhancement: The use of NLP techniques, as demonstrated by Kim et al. in NLPtoREST, provides a promising direction for improving REST API testing. By extracting additional rules from the human-readable parts of OpenAPI specifications, NLP-driven approaches can enhance the coverage and accuracy of automated testing tools. Future research should focus on refining these techniques to better handle the nuances of natural language, thereby improving the overall robustness of API testing methodologies. Automated test generation approaches such as RESTful API Automated Test Case Generation [56] and model-driven fuzzers like RESTler [57] have each shown promise in covering API state spaces efficiently [21].

Machine Learning Approaches: Machine learning model can be trained on large datasets of API interactions to predict and generate inputs that will be valid and typical. These models can be trained to generate inputs from past data and ensure the inputs are diverse and relevant to increase the chances of identifying intricate susceptibilities. Further, reinforcement learning techniques can be used to decide the input generation policies based on the results of the previous tests [58,59].

Fuzz Testing Enhancements: Fuzz testing, which aims to find software weaknesses by feeding it with random inputs, can be improved with the addition of domain knowledge and heuristics. Smart fuzzers that have knowledge of the API operations can create better test cases focusing on the areas that are more

susceptible to the attacks [20,60–62]. More recent hybrid fuzzers, e.g., MINER's data-driven approach [63], EDEFuzz's exposure-centric strategy [64], and KubeFuzzer for Kubernetes APIs [65], demonstrate even higher vulnerability discovery rates.

9.2 Improved Dependency Detection

Specifying and managing dependencies between API operations is one of the critical activities in stateful testing. Dependencies can exist when one API operation depends on the result of another operation or when several operations are to be performed in order to be effective.

Advanced Algorithms: It is imperative to develop complex algorithms that are able to analyze the sequence of API calls and determine dependencies. These algorithms should be able to recognize direct as well as indirect dependencies so that all the interactions are taken into account while testing. The relationships can be managed and visualized using graph based models and dependency graphs [32,66].

Machine Learning Techniques: Machine learning models can be trained to understand patterns and dependencies in API call sequences. These models can learn from historical API interactions to forecast and recognize dependencies to enhance the accuracy and coverage of stateful testing. In this context, techniques such as sequence-to-sequence models and recurrent neural networks (RNNs) can be especially effective [58,67,68].

Real-Time Monitoring: It is crucial to use tools that monitor API use and provide up-to-date dependency information in real time to improve the efficiency of testing. Such tools can monitor API usage in real-time and continuously update based on the evolution of dependencies. This will ensure that testing remains effective and thorough.

9.3 Comprehensive Specification Analysis

Addressing the discrepancies between API specifications and implementations is essential for accurate vulnerability detection. Specifications provide a formal description of the API's expected behavior, including endpoints, parameters, data types, and access controls.

Automated Specification Validation: Developing automated tools that can validate API specifications against actual implementations is crucial. These tools should identify discrepancies, highlight missing or incorrect information, and suggest corrections to ensure that specifications accurately reflect the API's behavior. Formal verification techniques, such as model checking and theorem proving, can be employed to validate the consistency and completeness of specifications [11–13].

Natural Language Processing (NLP) for Specifications: NLP techniques can be applied to analyze textual descriptions in API specifications and documentation. By extracting and understanding the intent and constraints described in natural language, these tools can cross-verify the specifications with the actual API behavior, ensuring that no critical information is overlooked [54,55].

Continuous Specification Analysis: Integrating specification analysis into the development workflow through Continuous Integration/Continuous Deployment (CI/CD) practices can ensure that specifications remain accurate and up-to-date. Automated tools can monitor changes in API specifications and implementations, providing real-time feedback to developers and preventing the introduction of discrepancies.

9.4 Real-World Validation

Validating tools and methodologies using real-world datasets and applications is essential to ensure their practical applicability and effectiveness. Controlled environments and synthetic benchmarks often fail to capture the complexities of real-world scenarios, leading to potential gaps in vulnerability detection.

Field Studies and Case Studies: Conducting large-scale field studies and case studies with industry partners can provide valuable insights into the performance of vulnerability detection tools in real-world settings. These studies should analyze how well the tools can detect vulnerabilities under various conditions, including different API load levels, response times, and integration complexities [10,32].

Collaborative Research: Establishing collaborations between academia and industry can facilitate access to diverse and representative API deployments. By working with real-world APIs, researchers can identify practical challenges, refine their methodologies, and develop solutions that are more relevant to industry needs.

Benchmarking with Real-World Data: Developing standardized benchmarks that incorporate real-world API datasets and scenarios can help evaluate the effectiveness of vulnerability detection tools. These benchmarks should reflect the diversity and complexity of modern API deployments, providing a realistic assessment of tool performance [2,12,13].

10 Conclusion

The synthesis and classification of the material analyzed in this paper emphasize the significance of addressing security vulnerabilities in RESTful APIs. As these APIs are becoming a more essential part of modern web services, the presence of various security threats is a considerable risk to the confidentiality, integrity, and availability of web services. Our proposed taxonomy divides these vulnerabilities into the following: authentication and authorization, data validation, and configuration and deployment. This classification will help in systematically addressing these risks.

Our review reveals several important areas that remain under-researched in the context of current developments in vulnerability detection tools and techniques. Most of the existing tools are evaluated in controlled settings and do not incorporate real-world conditions; hence, they do not cover all possible cases and can omit critical issues. The identification of dependency relationships between API operations still poses a problem, as do other issues such as the dynamic nature of APIs, which tend to change frequently and may be difficult to capture using current approaches. Furthermore, there is a need to conduct a detailed specification analysis to ensure that the APIs are implemented as intended to support their specifications in order to implement efficient security.

Closing these gaps calls for a comprehensive approach. Automated input generation methods based on natural language processing and machine learning can increase the coverage and realistic nature of generated test cases. Better algorithms for detecting dependencies and monitoring in real-time can make testing of API interactions more efficient and effective. Automated tools for continuous specification validation can help to fill the gap between specification of the API and the implementation of the security measures. Last but not least, field testing in real-world scenarios and partnerships with industry stakeholders are crucial to confirm the applicability of these tools and techniques.

In conclusion, further research on RESTful API vulnerability detection requires collective efforts of scholars and experts to enhance the development of more efficient security systems. Therefore, by increasing input generation, dependency detection, specification analysis, and real-world validation, researchers and practitioners can strengthen the resilience of RESTful APIs against emerging cyber threats. Our work offers a basis for future studies, paving the way for major developments in API security and supporting the continuous process of safeguarding the digital foundation of contemporary web applications.

Acknowledgement: The authors received no support.

Funding Statement: The authors received no specific funding for this study.

Author Contributions: Fatima Tanveer performed the literature search, conducted the data analysis, and drafted the manuscript for this review article. The initial idea for the review was proposed by Faisal Iradat and further refined by Waseem Iqbal. Awais Ahmad critically reviewed the manuscript and provided valuable feedback. All authors contributed to the critical revision of the manuscript. Faisal Iradat and Waseem Iqbal provided supervision and administration throughout the study. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: This article does not involve data availability, and this section is not applicable.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

1. Kornienko DV, Mishina SV, Shcherbatykh SV, Melnikov MO. Principles of securing RESTful API web services developed with python frameworks. J Phys Conf Ser. 2021;2094(3):032016. doi:10.1088/1742-6596/2094/3/032016.
2. Modi B, Chourasia U, Pandey R. Design and implementation of RESTFUL API based model for vulnerability detection and mitigation. IOP Conf Ser Mater Sci Eng. 2022;1228(1):012010. doi:10.1088/1757-899x/1228/1/012010.
3. Du W, Li J, Wang Y, Chen L, Zhao R, Zhu J, et al. Vulnerability-oriented Testing for RESTful APIs. In: 33rd USENIX Security Symposium (USENIX Security '24); 2024 Aug 14–16. Philadelphia, PA, USA. p. 739–55.
4. Zagane M, Abdi MK, Alenezi M. Deep learning for software vulnerabilities detection using code metrics. IEEE Access. 2020;8:74562–70. doi:10.1109/access.2020.2988557.
5. ur Rehman JA, Muhammad N, Alenezi M, Javed Y. Using public vulnerabilities data to self-heal security issues in software systems. ICIC Exp Lett. 2019;13(7):557.
6. Traceable AI. 2023 state of API security report. 2023 [Internet]. [cited 2025 Jun 23]. Available from: https://www.traceable.ai/2023-state-of-api-security.
7. Salt Security. API security trends 2024: the growing threat landscape. 2024 [Internet]. [cited 2025 Jun 23]. Available from: https://salt.security/blog/api-security-trends-2024.
8. Open Web Application Security Project (OWASP). API security top 10–2023. 2023 [Internet]. [cited 2025 Jun 23]. Available from: https://owasp.org/www-project-api-security.
9. OWASP Foundation. OWASP API security top 10–2023. 2023 [Internet]. [cited 2025 Jun 23]. Available from: https://owasp.org/API-Security/editions/2023/en/0x00-header/.
10. Golmohammadi A, Zhang M, Arcuri A. Testing RESTful APIs: a survey. arXiv:2212.14604. 2022.
11. Kim M, Xin Q, Sinha S, Orso A. Automated test generation for REST APIs: no time to rest yet. In: Proceedings of The 31st ACM SIGSOFT International Symposium on Software Testing and Analysis; 2022 Jul 18–22; Online. p. 289–301. doi:10.1145/3533767.3534401.
12. Barabanov A, Dergunov D, Makrushin D, Teplov A. Automatic detection of access control vulnerabilities via API specification processing. Voprosy Kiberbezopasnosti. 2022;1(47):49–65.
13. Deng G, Zhang Z, Li Y, Liu Y, Zhang T, Liu Y, et al. NAUTILUS: automated RESTful API vulnerability detection. In: Proceedings of the 32nd USENIX Conference on Security Symposium; 2023 Aug 9–11; Anaheim, CA, USA. p. 5593–609.
14. Wang Z, Tian W, Cui B. RESTlogic: detecting logic vulnerabilities in cloud REST APIs. Comput Mater Contin. 2024;78(2):1797–820. doi:10.32604/cmc.2023.047051.
15. Corradini D, Pasqua M, Ceccato M. Automated black-box testing of mass assignment vulnerabilities in RESTful APIs. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE); 2023 May 14–20; Melbourne, VIC, Australia. p. 2553–64.
16. Mazidi A, Corradini D, Ghafari M. Mining REST APIs for potential mass assignment vulnerabilities. In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering; 2024 Jun 18–21; Salerno, Italy. p. 369–74.

17. Fadlalla F, Elshoush F, Huwaida T. Input validation vulnerabilities in web applications: systematic review, classification, and analysis of the current state-of-the-art. IEEE Access. 2023;11(3):40128–61. doi:10.1109/access.2023.3266385.
18. Yu F, Alkhalaf M, Bultan T. Patching vulnerabilities with sanitization synthesis. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE); 2011 May 21–28; Honolulu, HI, USA. p. 251–60.
19. Barlas E, Du X, Davis JC. Exploiting input sanitization for regex denial of service. arXiv:2303.01996. 2023.
20. Wang J, Zhang Q, Rong H, Xu GH, Kim M. Leveraging hardware probes and optimizations for accelerating fuzz testing of heterogeneous applications. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE); 2023 Dec 3–9; San Francisco, CA, USA. p. 1101–13.
21. Kim M, Corradini D, Sinha S, Orso A, Pasqua M, Tzoref-Brill R, et al. Enhancing REST API Testing with NLP Techniques. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis; 2023 Jul 17–21; Seattle, WA, USA. p. 1232–43.
22. Liao S, Cheng L, Luo X, Song Z, Cai H, Yao D, et al. A first look at security and privacy risks in the RapidAPI ecosystem. In: CCS '24: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security; 2024 Oct 14–18; Salt Lake City, UT, USA. p. 1626–40.
23. Iqbal W, Abbas H, Daneshmand M, Rauf B, Abbas Y. An in-depth analysis of IoT security requirements, challenges, and their countermeasures via software-defined security. IEEE Internet Things J. 2020;7(10):10250–76. doi:10.1109/jiot.2020.2997651.
24. Mousavi Z, Islam C, Babar MA, Abuadbba A, Moore K. Detecting misuse of security APIs: a systematic review. arXiv:2306.08869. 2024.
25. Acunetix. API Security risks and challenges; Acunetix web security report . 2024 [Internet]. [cited 2025 Jun 23]. Available from: https://www.acunetix.com/blog/articles/api-security-risks.
26. Rsnake. XSS cheat sheet. 2008 [Internet]. [cited 2025 Jun 23]. Available from: http://ha.ckers.org/xss.html.
27. HTML5 Security Team. HTML5 security cheat sheet [Internet]. [cited 2025 Jun 23]. Available from: http://html5sec.org/.
28. PortSwigger Web Security Team. Cross-site scripting (XSS) cheat sheet [Internet]. [cited 2025 Jun 23]. Available from: https://portswigger.net/web-security/cross-site-scripting/cheat-sheet.
29. XssPayloads. XSS payloads on X (Twitter) [Internet]. [cited 2025 Jun 23]. Available from: https://twitter.com/xsspayloads.
30. Kim M, Sinha S, Orso A. Adaptive REST API testing with reinforcement learning. arXiv:2309.04583. 2023.
31. Wang Y, Xu Y. Beyond REST: introducing APIF for comprehensive API vulnerability fuzzing. In: The 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024); 2024 Sep 30–Oct 2; Padua, Italy.
32. Liu Y, Li Y, Deng G, Liu Y, Wan R, Wu R, et al. Morest: model-based RESTful API testing with execution feedback. In: Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE); 2022 May 25–27; Pittsburgh, PA, USA. p. 1406–17.
33. Zhang Y, Wang X, Luo Q, Liu Q. Cross-site scripting attacks in social network APIs. In: Workshop on WEB 2.0 Security and Privacy (W2SP); 2013 May 23–24; San Francisco, CA, USA.
34. Project G. Stored cross-site scripting (XSS) vulnerability in REST resources API. 2024 [Internet]. [cited 2025 Jun 23]. Available from: https://github.com/geoserver/geoserver/security/advisories/GHSA-fh7p-5f6g-vj2w.
35. Tariq U, Ahmed I, Bashir AK, Shaukat K. A critical cybersecurity analysis and future research directions for the internet of things: a comprehensive review. Sensors. 2023;23(8):8. doi:10.3390/s23084117.
36. Kumar R, Godishala AK, Malaga RR, Kagitapu P. Securing web apps: analysis to understand common vulnerabilities, attack scenarios, and protective measures. In: ICCDE '24: Proceedings of the 2024 10th International Conference on Computing and Data Engineering; 2024 Jan 15–17; Bangkok, Thailand. p. 64–70.
37. Dias T, Maia E, Praça I. FuzzTheREST: an intelligent automated black-box RESTful API fuzzer. arXiv:2407.14361. 2024.

38. Kong Inc. API Impact Report 2024: AI edition—innovation and adoption challenges . 2024 [Internet]. [cited 2025 Jun 23]. Available from: https://konghq.com.
39. Garimilla M. The art of API design: best practices for modern software development. Int J Eng Tech Res (IJETR). 2024;9(2):229–39.
40. Kumar S, Kaur A, Jolly A, Baz M, Cheikhrouhou O. Cost benefit analysis of incorporating security and evaluation of its effects on various phases of agile software development. Math Probl Eng. 2021;2021:1–10. doi:10.1155/2021/7837153.
41. Alenezi M, Zagane M, Javed Y. Efficient deep features learning for vulnerability detection using character n-gram embedding. Jordanian J Comput Inf Technol (JJCIT). 2021;7(1):25–38. doi:10.5455/jjcit.71-1597824949.
42. Hadi HJ, Adnan M, Cao Y, Hussain FB, Ahmad N, Alshara MA, et al. iKern: advanced intrusion detection and prevention at the kernel level using eBPF. Technologies. 2024;12(8):122. doi:10.3390/technologies12080122.
43. Vulnerability Lab. Technical attack sheet for cross-site penetration tests [Internet]. [cited 2025 Jun 23]. Available from: http://www.vulnerability-lab.com/resources/documents/531.txt.
44. Yerushalmi S. GraphQL vulnerabilities and common attacks: what you need to know; imperva blog [Internet]. 2023 [cited 2025 Jun 23]. Available from: https://www.imperva.com/blog/graphql-vulnerabilities-common-attacks/.
45. Amareen S, Soto Dector O, Dado A, Bosu A. GraphQL adoption and challenges: community-driven insights from stackoverflow discussions. arXiv:2408.08363. 2024.
46. Pan H, Fang Y, Guo W, Xu Y, Wang C. Few-shot graph classification on cross-site scripting attacks detection. Comput Secur. 2024;140:103749.
47. Kim M, Sinha S, Orso A. LlamaRestTest: effective REST API testing with small language models. arXiv:2501.08598. 2025.
48. Jones M, Bayesh M, Jahan S. Unlocking deeper understanding: leveraging explainable AI for API anomaly detection insights. In: ICMLC '24: Proceedings of the 2024 16th International Conference on Machine Learning and Computing; 2024 Feb 2–5; Shenzhen, China. p. 211–7.
49. Technologies A. API security impact study 2025: the costs of API attacks in 4 APAC Countries; akamai technologies [Internet]. 2025 [cited 2025 Jun 23]. Available from: https://www.akamai.com.
50. Bogner J, Kotstein S, Pfaff T. Do RESTful API design rules have an impact on the understandability of web APIs? Empir Softw Eng. 2023;28:132. doi:10.1007/s10664-023-10367-y.
51. Kim M, Stennett T, Shah D, Sinha S, Orso A. Leveraging large language models to improve REST API testing. In: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results; 2024 April 14–20; Lisbon, Portugal. p. 37–41.
52. McFadden S, Maugeri M, Hicks C, Mavroudis V, Pierazzi F. WENDIGO: deep reinforcement learning for denial-of-service query discovery in GraphQL. In: 2024 IEEE Security & Privacy Workshops (SPW); 2024 May 23; San Francisco, CA, USA. p. 68–75. doi:10.1109/spw63631.2024.00012.
53. Santos Filho A, Rodríguez RJ, Feitosa EL. Automated broken object-level authorization attack detection in REST APIs through OpenAPI to colored petri nets transformation. Int J Inf Secur. 2025;24(2):83. doi:10.1007/s10207-024-00970-5.
54. Wolf T, Debut L, Sanh V, Chaumond J, Delangue C, Moi A, et al. HuggingFace's transformers: state-of-the-art natural language processing. arXiv:191003771. 2019.
55. Manning C, Surdeanu M, Bauer J, Finkel J, Bethard S, McClosky D. The stanford CoreNLP natural language processing toolkit. In: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations; 2014 Jun 22–27; Baltimore, MD, USA. p. 55–60.
56. Arcuri A. RESTful API automated test case generation. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS); 2017 Jul 25–29; Prague, Czech Republic. p. 9–20.
57. Atlidakis V, Godefroid P, Polishchuk M. RESTler: stateful REST API fuzzing. In: Proceedings of the 41st International Conference on Software Engineering (ICSE); 2019 May 27; Montreal, QC, Canada. p. 748–58.
58. Namita, Prachi, Sharma P. Windows malware detection using machine learning and TF-IDF enriched API calls information. In: 2022 Second International Conference on Computer Science, Engineering and Applications (ICCSEA); 2022 Sep 8; Gunupur, India. p. 1–6.

59. Dezfouli MP. Automated real-time machine learning for IoT for manufacturing a cloud architecture and API; 2020. [cited 2025 Jun 23]. Available from: https://hdl.handle.net/1853/62335.
60. Vinzenz N, Oka DK. Processing fuzz testing results into an evidence report. Warrendale, PA, USA: SAE; 2023. Paper #2023-01–0039. doi:10.4271/2023-01-0039.
61. Yang J, Arya S, Wang Y. Formal-Guided fuzz testing: targeting security assurance from specification to implementation for 5G and beyond. IEEE Access. 2024;12:29175–93. doi:10.1109/access.2024.3369613.
62. Touqir A, Iradat F, Iqbal W, Rakib A, Taskin N, Jadidbonab H, et al. Systematic exploration of fuzzing in IoT: techniques, vulnerabilities, and open challenges. J Supercomput. 2025;81(8):877. doi:10.1007/s11227-025-07371-y.
63. Lyu C, Xu J, Ji S, Zhang X, Wang Q, Zhao B, et al. MINER: a hybrid data-driven approach for rEST API fuzzing. In: 32nd USENIX Security Symposium (USENIX Security'23); 2023 Aug 9–11; Anaheim, CA, USA. p. 4517–34.
64. Pan L, Cohney S, Murray T, Pham VT. EDEFuzz: A web API fuzzer for excessive data exposures. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering; 2024 Apr 14–20; Lisbon, Portugal.
65. Zheng T, Tang R, Chen X, Shen C. KubeFuzzer: automating RESTful API vulnerability detection in kubernetes. Comput Mater Contin. 2024;81(1):1595–612. doi:10.32604/cmc.2024.055180.
66. Czemerinski H, Braberman V, Uchitel S. Behavior abstraction adequacy criteria for API call protocol testing. Softw Testing Verification Reliab. 2016;26(3):211–44. doi:10.1002/stvr.1593.
67. Konstas I, Iyer S, Yatskar M, Choi Y, Zettlemoyer L. Neural AMR: sequence-to-sequence models for parsing and generation. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics; 2017 Jul 30–Aug 4; Vancouver, BC, Canada. p. 146–57.
68. Lewis M, Liu Y, Goyal N, Ghazvininejad M, Mohamed AR, Levy O, et al. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proceedings of the Annual Meeting of the Association for Computational Linguistics; 2019 Jul 28–Aug 2; Florence, Italy. p. 7871–80.